SCons Design version 0.91

Steven Knight

SCons Design version 0.91
by Steven Knight

Revision 0.01.D177 (2001/12/13 20:55:46) Edition
Published 2001
Copyright © 2001 Steven Knight

Copyright (c) 2001 Steven Knight Portions of this document, by the same author, were previously published Copyright 2000 by CodeSourcery LLC, under the Software Carpentry Open Publication License, the terms of which are available at http://www.software-carpentry.com/openpub-license.html [1].

Table of Contents

Chapter 1. Introduction

The `SCons` tool provides an easy-to-use, feature-rich interface for constructing software. Architecturally, `SCons` separates its dependency analysis and external object management into an interface-independent Build Engine that could be embedded in any software system that can run Python.

At the command line, `SCons` presents an easily-grasped tool where configuration files are Python scripts, reducing the need to learn new build-tool syntax. Inexperienced users can use intelligent methods that "do the right thing" to build software with a minimum of fuss. Sophisticated users can use a rich set of underlying features for finer control of the build process, including mechanisms for easily extending the build process to new file types.

Dependencies are tracked using digital signatures, which provide more robust dependency analysis than file time stamps. Implicit dependencies are determined automatically by scanning the contents of source files, avoiding the need for laborious and fragile maintenance of static lists of dependencies in configuration files.

The `SCons` tool supports use of files from one or more central code repositories, a mechanism for caching derived files, and parallel builds. The tool also includes a framework for sharing build environments, which allows system administrators or integrators to define appropriate build parameters for use by other users.

About This Document

This document is an ongoing work-in-progress to write down the ideas and tradeoffs that have gone, and will go into, the `SCons` design. As such, this is intended primarily for use by developers and others working on `SCons`, although it is also intended to serve as a detailed overview of `SCons` for other interested parties. It will be continually updated and evolve, and will likely overlap with other documentation produced by the project. Sections of this document that deal with syntax, for example, may move or be copied into a user guide or reference manual.

So please don't assume that everything mentioned here has been decided and carved in stone. If you have ideas for improvements, or questions about things that don't seem to make any sense, please help improve the design by speaking up about them.

* *Sections marked like this (prefixed with `RATIONALE:` in the HTML, surrounded by `BEGIN RATIONALE:` and `END RATIONALE:` in the printed documentatio) are DocBook REMARKs, comments about the document rather than actual document. I've used these to mark sections that need work, but also to cite some open design issues. If you have input on any of these marked issues, I'm especially eager to hear it.*

Chapter 2. Goals

As a next-generation build tool, `SCons` should fundamentally improve on its predecessors. Rather than simply being driven by trying to *not* be like previous tools, `SCons` aims to satisfy the following goals:

`Practicality`

The `SCons` design emphasizes an implementable feature set that lets users get practical, useful work done. `SCons` is helped in this regard by its roots in `Cons`, which has had its feature set honed by several years of input from a dedicated band of users.

`Portability`

`SCons` is intended as a portable build tool, able to handle software construction tasks on a variety of operating systems. It should be possible (although not mandatory) to use `SCons` so that the same configuration file builds the same software correctly on, for example, both Linux and Windows NT. Consequently, `SCons` should hide from users operating-system-dependent details such as filename extensions (for example, `.o` vs. `.obj`).

`Usability`

Novice users should be able to grasp quickly the rudiments of using `SCons` to build their software. This extends to installing `SCons`, too. Installation should be painless, and the installed `SCons` should work "out of the box" to build most software.

This goal should be kept in mind during implementation, when there is always a tendency to try to optimize too early. Speed is nice, but not as important as clarity and ease of use.

`Utility`

`SCons` should also provide a rich enough set of features to accommodate building more complicated software projects. However, the features required for building complicated software projects should not get in the way of novice users. (See the previous goal.) In other words, complexity should be available when it's needed but not required to get work done. Practically, this implies that `SCons` shouldn't be dumbed down to the point it excludes complicated software builds.

`Sharability`

As a key element in balancing the conflicting needs of `Usability` and `Utility`, `SCons` should provide mechanisms to allow `SCons` users to share build rules, dependency scanners, and other objects and recipes for constructing software. A good sharing mechanism should support the model wherein most developers on a project use rules and templates that are created and maintained by a local integrator or build-master,

`Extensibility`

`SCons` should provide mechanisms for easily extending its capabilities, including building new types of files, adding new types of dependency scanning, being able to accomodate dependencies between objects other than files, etc.

`Flexibility`

In addition to providing a useful command-line interface, `SCons` should provide the right architectural framework for embedding its dependency management in other interfaces. `SCons` would help strengthen other GUIs or IDEs and the additional requirements of the other interfaces would help broaden and solidify the core `SCons` dependency management.

Fixing `Make`'s problems

* *To be written.*

Fixing `Cons`'s problems

* *To be written.*

Chapter 3. Overview

Architecture

The heart of `SCons` is its *Build Engine*. The `SCons` Build Engine is a Python module that manages dependencies between external objects such as files or database records. The Build Engine is designed to be interface-neutral and easily embeddable in any software system that needs dependency analysis between updatable objects.

The key parts of the Build Engine architecture are captured in the following quasi-UML diagram:

* *Including this figure makes our PDF build blow up. The figure, however, is left over from the Software Carpentry contest and is therefore old, out-of-date, and needs to be redone anyway. This is where it will go, anyway...*

The point of `SCons` is to manage dependencies between arbitrary external objects. Consequently, the Build Engine does not restrict or specify the nature of the external objects it manages, but instead relies on subclass of the `Node` class to interact with the external system or systems (file systems, database management systems) that maintain the objects being examined or updated.

The Build Engine presents to the software system in which it is embedded a Python API for specifying source (input) and target (output) objects, rules for building/updating objects, rules for scanning objects for dependencies, etc. Above its Python API, the Build Engine is completely interface-independent, and can be encapsulated by any other software that supports embedded Python.

Software that chooses to use the Build Engine for dependency management interacts with it through *Construction Environments*. A Construction Environment consists of a dictionary of environment variables, and one or more associated `Scanner` objects and `Builder` objects. The Python API is used to form these associations.

A `Scanner` object specifies how to examine a type of source object (C source file, database record) for dependency information. A `Scanner` object may use variables from the associated Construction Environment to modify how it scans an object: specifying a search path for included files, which field in a database record to consult, etc.

A `Builder` object specifies how to update a type of target object: executable program, object file, database field, etc. Like a `Scanner` object, a `Builder` object may use variables from the associated Construction Environment to modify how it builds an object: specifying flags to a compiler, using a different update function, etc.

`Scanner` and `Builder` objects will return one or more `Node` objects that represent external objects. `Node` objects are the means by which the Build Engine tracks dependencies: A `Node` may represent a source (input) object that should already exist, or a target (output) object which may be built, or both. The `Node` class is sub-classed to represent external objects of specific type: files, directories, database fields or records, etc. Because dependency information, however, is tracked by the top-level `Node` methods and attributes, dependencies can exist between nodes representing different external object types. For example, building a file could be made dependent on the value of a given field in a database record, or a database table could depend on the contents of an external file.

The Build Engine uses a `Job` class (not displayed) to manage the actual work of updating external target objects: spawning commands to build files, submitting the necessary commands to update a database record, etc. The `Job` class has sub-classes to handle differences between spawning jobs in parallel and serially.

The Build Engine also uses a `Signature` class (not displayed) to maintain information about whether an external object is up-to-date. Target objects with out-of-date signatures are updated using the appropriate `Builder` object.

Build Engine

More detailed discussion of some of the Build Engine's characteristics:

Python API

The Build Engine can be embedded in any other software that supports embedding Python: in a GUI, in a wrapper script that interprets classic `Makefile` syntax, or in any other software that can translate its dependency representation into the appropriate calls to the Build Engine API. describes in detail the specification for a "Native Python" interface that will drive the `SCons` implementation effort.

Single-image execution

When building/updating the objects, the Build Engine operates as a single executable with a complete Directed Acyclic Graph (DAG) of the dependencies in the entire build tree. This is in stark contrast to the commonplace recursive use of Make to handle hierarchical directory-tree builds.

Dependency analysis

Dependency analysis is carried out via digital signatures (a.k.a. "fingerprints"). Contents of object are examined and reduced to a number that can be stored and compared to see if the object has changed. Additionally, `SCons` uses the same signature technique on the command-lines that are executed to update an object. If the command-line has changed since the last time, then the object must be rebuilt.

Customized output

The output of Build Engine is customizable through user-defined functions. This could be used to print additional desired information about what `SCons` is doing, or tailor output to a specific build analyzer, GUI, or IDE.

Build failures

`SCons` detects build failures via the exit status from the tools used to build the target files. By default, a failed exit status (non-zero on UNIX systems) terminates the build with an appropriate error message. An appropriate class from the Python library will interpret build-tool failures via an OS-independent API.

If multiple tasks are executing in a parallel build, and one tool returns failure, `SCons` will not initiate any further build tasks, but allow the other build tasks to complete before terminating.

A `-k` command-line option may be used to ignore errors and continue building other targets. In no case will a target that depends on a failed build be rebuilt.

Interfaces

As previously described, the `SCons` Build Engine is interface-independent above its Python API, and can be embedded in any software system that can translate its dependency requirements into the necessary Python calls.

The "main" `SCons` interface for implementation purposes, uses Python scripts as configuration files. Because this exposes the Build Engine's Python API to the user, it is current called the "Native Python" interface.

This section will also discuss how `SCons` will function in the context of two other interfaces: the `Makefile` interface of the classic `Make` utility, and a hypothetical graphical user interface (GUI).

Native Python interface

The Native Python interface is intended to be the primary interface by which users will know `SCons`--that is, it is the interface they will use if they actually type `SCons` at a command-line prompt.

In the Native Python interface, `SCons` configuration files are simply Python scripts that directly invoke methods from the Build Engine's Python API to specify target files to be built, rules for building the target files, and dependencies. Additional methods, specific to this interface, are added to handle functionality that is specific to the Native Python interface: reading a subsidiary configuration file; copying target files to an installation directory; etc.

Because configuration files are Python scripts, Python flow control can be used to provide very flexible manipulation of objects and dependencies. For example, a function could be used to invoke a common set of methods on a file, and called iteratively over an array of files.

As an additional advantage, syntax errors in `SCons` Native Python configuration files will be caught by the Python parser. Target-building does not begin until after all configuration files are read, so a syntax error will not cause a build to fail half-way.

Makefile interface

An alternate `SCons` interface would provide backwards compatibility with the classic `Make` utility. This would be done by embedding the `SCons` Build Engine in a Python script that can translate existing `Makefile`s into the underlying calls to the Build Engine's Python API for building and tracking dependencies. Here are approaches to solving some of the issues that arise from marrying these two pieces:

- `Makefile` suffix rules can be translated into an appropriate `Builder` object with suffix maps from the Construction Environment.
- Long lists of static dependences appended to a `Makefile` by various **"make depend"** schemes can be preserved but supplemented by the more accurate dependency information provided by `Scanner` objects.
- Recursive invocations of `Make` can be avoided by reading up the subsidiary `Makefile` instead.

Lest this seem like too outlandish an undertaking, there is a working example of this approach: Gary Holt's `Make++` utility is a Perl script that provides admirably complete parsing of complicated `Makefile`s around an internal build engine inspired, in part, by the classic `Cons` utility.

Graphical interfaces

The `SCons` Build Engine is designed from the ground up to be embedded into multiple interfaces. Consequently, embedding the dependency capabilities of `SCons` into graphical interface would be a matter of mapping the GUI's dependency representation (either implicit or explicit) into corresponding calls to the Python API of the `SCons` Build Engine.

Note, however, that this proposal leaves the problem of designed a good graphical interface for representing software build dependencies to people with actual GUI design experience...

Chapter 4. Build Engine API

General Principles

Keyword arguments

All methods and functions in this API will support the use of keyword arguments in calls, for the sake of explicitness and readability. For brevity in the hands of experts, most methods and functions will also support positional arguments for their most-commonly-used arguments. As an explicit example, the following two lines will each arrange for an executable program named `foo` (or `foo.exe` on a Win32 system) to be compiled from the `foo.c` source file:

```
env.Program(target = 'foo', source = 'foo.c')

env.Program('foo', 'foo.c')
```

Internal object representation

All methods and functions use internal (Python) objects that represent the external objects (files, for example) for which they perform dependency analysis.

All methods and functions in this API that accept an external object as an argument will accept *either* a string description or an object reference. For example, the two following two-line examples are equivalent:

```
env.Object(target = 'foo.o', source = 'foo.c')
env.Program(target = 'foo', 'foo.o')    # builds foo from foo.o

foo_obj = env.Object(target = 'foo.o', source = 'foo.c')
env.Program(target = 'foo', foo_obj)    # builds foo from foo.o
```

Construction Environments

A `construction environment` is the basic means by which a software system interacts with the `SCons` Python API to control a build process.

A `construction environment` is an object with associated methods for generating target files of various types (`Builder` objects), other associated object methods for automatically determining dependencies from the contents of various types of source files (`Scanner` objects), and a dictionary of values used by these methods.

Passing no arguments to the `Environment` instantiation creates a `construction environment` with default values for the current platform:

```
env = Environment()
```

Construction variables

A `construction environment` has an associated dictionary of `construction variables` that control how the build is performed. By default, the `Environment` method creates a `construction environment` with values that make most software build "out of the box" on the host system. These default values will be generated at the time SCons is installed using functionality similar to that provided by GNU Autoconf. [1] At a minimum, there will be pre-configured sets of default values that will provide reasonable defaults for UNIX and Windows NT.

The default construction environment values may be overridden when a new construction environment is created by specifying keyword arguments:

```
env = Environment(CC =          'gcc',
                  CCFLAGS =    '-g',
                  CPPPATH =    ['.', 'src', '/usr/include'],
                  LIBPATH =    ['/usr/lib', '.'])
```

Fetching construction variables

A copy of the dictionary of construction variables can be returned using the Dictionary method:

```
env = Environment()
dict = env.Dictionary()
```

If any arguments are supplied, then just the corresponding value(s) are returned:

```
ccflags = env.Dictionary('CCFLAGS')
cc, ld = env.Dictionary('CC', 'LD')
```

Copying a construction environment

A method exists to return a copy of an existing environment, with any overridden values specified as keyword arguments to the method:

```
env = Environment()
debug = env.Copy(CCFLAGS = '-g')
```

Multiple construction environments

Different external objects often require different build characteristics. Multiple construction environments may be defined, each with different values:

```
env = Environment(CCFLAGS = '')
debug = Environment(CCFLAGS = '-g')
env.Make(target = 'hello', source = 'hello.c')
debug.Make(target = 'hello-debug', source = 'hello.c')
```

Dictionaries of values from multiple construction environments may be passed to the Environment instantiation or the Copy method, in which case the last-specified dictionary value wins:

```
env1 = Environment(CCFLAGS = '-O', LDFLAGS = '-d')
env2 = Environment(CCFLAGS = '-g')
new = Environment(env1.Dictionary(), env2.Dictionary())
```

The new environment in the above example retains LDFLAGS = '-d' from the env1 environment, and CCFLAGS = '-g' from the env2 environment.

Variable substitution

Within a construction command, any variable from the `construction environment` may be interpolated by prefixing the name of the construction with $:

```
MyBuilder = Builder(command = "$XX $XXFLAGS -c $_INPUTS -o $target")

env.Command(targets = 'bar.out', sources = 'bar.in',
            command = "sed '1d' < $source > $target")
```

Variable substitution is recursive: the command line is expanded until no more substitutions can be made.

Variable names following the $ may be enclosed in braces. This can be used to concatenate an interpolated value with an alphanumeric character:

```
VerboseBuilder = Builder(command = "$XX -${XXFLAGS}v > $target")
```

The variable within braces may contain a pair of parentheses after a Python function name to be evaluated (for example, `${map()}`). `SCons` will interpolate the return value from the function (presumably a string):

```
env = Environment(FUNC = myfunc)
env.Command(target = 'foo.out', source = 'foo.in',
            command = "${FUNC($<)}")
```

If a referenced variable is not defined in the `construction environment`, the null string is interpolated.

The following special variables can also be used:

`$targets`

All target file names. If multiple targets are specified in an array, `$targets` expands to the entire list of targets, separated by a single space.

Individual targets from a list may be extracted by enclosing the `targets` keyword in braces and using the appropriate Python array index or slice:

```
${targets[0]}     # expands to the first target

${targets[1:]}    # expands to all but the first target

${targets[1:-1]}  # expands to all but the first and last targets
```

`$target`

A synonym for `${targets[0]}`, the first target specified.

`$sources`

All input file names. Any input file names that are used anywhere else on the current command line (via `${sources[0]}`, `${sources{[1]}`, etc.) are removed from the expanded list.

Any of the above special variables may be enclosed in braces and followed immediately by one of the following attributes to select just a portion of the expanded path name:

`.base`

Basename: the directory plus the file name, minus any file suffix.

`.dir`

The directory in which the file lives. This is a relative path, where appropriate.

`.file`

The file name, minus any directory portion.

`.suffix`

The file name suffix (that is, the right-most dot in the file name, and all characters to the right of that).

`.filebase`

The file name (no directory portion), minus any file suffix.

`.abspath`

The absolute path to the file.

`Builder` Objects

By default, `SCons` supplies (and uses) a number of pre-defined `Builder` objects:

`Object`	compile or assemble an object file
`Library`	archive files into a library
`SharedLibrary`	archive files into a shared library
`Program`	link objects and/or libraries into an executable
`Make`	build according to file suffixes; see below

* *`Library` and `SharedLibrary` have nearly identical semantics, just different tools and `construction environments` (paths, etc.) that they use. In other words, you can construct a shared library using just the `Library Builder` object with a different environment. I think that's a better way to do it. Feedback?*

A `construction environment` can be explicitly initialized with associated `Builder` objects that will be bound to the `construction environment` object:

```
env = Environment(BUILDERS = ['Object', 'Program'])
```

`Builder` objects bound to a `construction environment` can be called directly as methods. When invoked, a `Builder` object returns a (list of) objects that it will build:

```
obj = env.Object(target ='hello.o', source = 'hello.c')
lib = env.Library(target ='libfoo.a',
                  source = ['aaa.c', 'bbb.c'])
slib = env.SharedLibrary(target ='libbar.so',
                         source = ['xxx.c', 'yyy.c'])
prog = env.Program(target ='hello',
                   source = ['hello.o', 'libfoo.a', 'libbar.so'])
```

Specifying multiple inputs

Multiple input files that go into creating a target file may be passed in as a single string, with the individual file names separated by white space:

```
env.Library(target = 'foo.a', source = 'aaa.c bbb.c ccc.c')
env.Object(target = 'yyy.o', source = 'yyy.c')
env.Program(target = 'bar', source = 'xxx.c yyy.o foo.a')
```

Alternatively, multiple input files that go into creating a target file may be passed in as an array. This allows input files to be specified using their object representation:

```
env.Library(target = 'foo.a', source = ['aaa.c', 'bbb.c', 'ccc.c'])
yyy_obj = env.Object(target = 'yyy.o', source = 'yyy.c')
env.Program(target = 'bar', source = ['xxx.c', yyy_obj, 'foo.a'])
```

Individual string elements within an array of input files are *not* further split into white-space separated file names. This allows file names that contain white space to be specified by putting the value into an array:

```
env.Program(target = 'foo', source = ['an input file.c'])
```

Specifying multiple targets

Conversely, the generated target may be a string listing multiple files separated by white space:

```
env.Object(target = 'grammar.o y.tab.h', source = 'grammar.y')
```

An array of multiple target files can be used to mix string and object representations, or to accomodate file names that contain white space:

```
env.Program(target = ['my program'], source = 'input.c')
```

File prefixes and suffixes

For portability, if the target file name does not already have an appropriate file prefix or suffix, the `Builder` objects will append one appropriate for the file type on the current system:

```
# builds 'hello.o' on UNIX, 'hello.obj' on Windows NT:
obj = env.Object(target ='hello', source = 'hello.c')

# builds 'libfoo.a' on UNIX, 'foo.lib' on Windows NT:
lib = env.Library(target ='foo', source = ['aaa.c', 'bbb.c'])

# builds 'libbar.so' on UNIX, 'bar.dll' on Windows NT:
slib = env.SharedLibrary(target ='bar', source = ['xxx.c', 'yyy.c'])

# builds 'hello' on UNIX, 'hello.exe' on Windows NT:
prog = env.Program(target ='hello',
                   source = ['hello.o', 'libfoo.a', 'libbar.so'])
```

`Builder` object exceptions

`Builder` objects raise the following exceptions on error:

* *LIST THESE ONCE WE FIGURE OUT WHAT THEY ARE FROM CODING THEM.*

User-defined `Builder` objects

Users can define additional `Builder` objects for specific external object types unknown to `SCons`. A `Builder` object may build its target by executing an external command:

```
WebPage = Builder(command = 'htmlgen $HTMLGENFLAGS $sources > $target',
                  suffix = '.html',
                  src_suffix = '.in')
```

Alternatively, a `Builder` object may also build its target by executing a Python function:

```
def update(dest):
        # [code to update the object]
        return 1

OtherBuilder1 = Builder(function = update,
                        src_suffix = ['.in', '.input'])
```

An optional argument to pass to the function may be specified:

```
def update_arg(dest, arg):
        # [code to update the object]
        return 1

OtherBuilder2 = Builder(function = update_arg,
                        function_arg = 'xyzzy',
                        src_suffix = ['.in', '.input'])
```

Both an external command and an internal function may be specified, in which case the function will be called to build the object first, followed by the command line.

* *NEED AN EXAMPLE HERE.*

User-defined `Builder` objects can be used like the default `Builder` objects to initialize `construction environments`.

```
WebPage = Builder(command = 'htmlgen $HTMLGENFLAGS $sources > $target',
                  suffix = '.html',
                  src_suffix = '.in')
env = Environment(BUILDERS = ['WebPage'])
env.WebPage(target = 'foo.html', source = 'foo.in')
# Builds 'bar.html' on UNIX, 'bar.htm' on Windows NT:
env.WebPage(target = 'bar', source = 'bar.in')
```

The command-line specification can interpolate variables from the `construction environment`; see "Variable substitution," above.

A `Builder` object may optionally be initialized with a list of:

- the prefix of the target file (e.g., 'lib' for libraries)
- the suffix of the target file (e.g., '.a' for libraries)

- the expected suffixes of the input files (e.g., '.o' for object files)

These arguments are used in automatic dependency analysis and to generate output file names that don't have suffixes supplied explicitly.

Copying `Builder` Objects

A `Copy` method exists to return a copy of an existing `Builder` object, with any overridden values specified as keyword arguments to the method:

```
build = Builder(function = my_build)
build_out = build.Copy(suffix = '.out')
```

Typically, `Builder` objects will be supplied by a tool-master or administrator through a shared `construction environment`.

Special-purpose build rules

A pre-defined `Command` builder exists to associate a target file with a specific command or list of commands for building the file:

```
env.Command(target = 'foo.out', source =
            command = 'foo.in', "foo.process $sources > $target")

commands = [    "bar.process -o .tmpfile $sources",
                "mv .tmpfile $target" ]
env.Command(target = 'bar.out', source = 'bar.in', command = commands)
```

This is useful when it's too cumbersome to create a `Builder` object just to build a single file in a special way.

The `Make Builder`

A pre-defined `Builder` object named `Make` exists to make simple builds as easy as possible for users, at the expense of sacrificing some build portability.

The following minimal example builds the 'hello' program from the 'hello.c' source file:

```
Environment().Make('hello', 'hello.c')
```

Users of the `Make Builder` object are not required to understand intermediate steps involved in generating a file--for example, the distinction between compiling source code into an object file, and then linking object files into an executable. The details of intermediate steps are handled by the invoked method. Users that need to, however, can specify intermediate steps explicitly:

```
env = Environment()
env.Make(target = 'hello.o', source = 'hello.c')
env.Make(target = 'hello', source = 'hello.o')
```

The `Make` method understands the file suffixes specified and "does the right thing" to generate the target object and program files, respectively. It does this by examining the specified output suffixes for the `Builder` objects bound to the environment.

Because file name suffixes in the target and source file names must be specified, the `Make` method can't be used portably across operating systems. In other words, for the example above, the `Make` builder will not generate `hello.exe` on Windows NT.

`Builder` maps

* *Do we even need this anymore? Now that the individual builders have specified `suffix` and `src_suffix` values, all of the information we need to support the `Make` builder is right there in the environment. I think this is a holdover from before I added the `suffix` arguments. If you want `Make` to do something different, you set it up with another environment...*

The `env.Make` method "does the right thing" to build different file types because it uses a dictionary from the `construction environment` that maps file suffixes to the appropriate `Builder` object. This `BUILDERMAP` can be initialized at instantiation:

```
env = Environment(BUILDERMAP = {
                        '.o' : Object,
                        '.a' : Library,
                        '.html' : WebPage,
                        '' : Program,
                })
```

With the `BUILDERMAP` properly initialized, the `env.Make` method can be used to build additional file types:

```
env.Make(target = 'index.html', source = 'index.input')
```

`Builder` objects referenced in the `BUILDERMAP` do not need to be listed separately in the `BUILDERS` variable. The `construction environment` will bind the union of the `Builder` objects listed in both variables.

Dependencies

Automatic dependencies

By default, `SCons` assumes that a target file has `automatic dependencies` on the:

- tool used to build the target file
- contents of the input files
- command line used to build the target file

If any of these changes, the target file will be rebuilt.

Implicit dependencies

Additionally, `SCons` can scan the contents of files for `implicit dependencies` on other files. For example, `SCons` will scan the contents of a `.c` file and determine that any object created from it is dependent on any `.h` files specified via `#include`. `SCons`, therefore, "does the right thing" without needing to have these dependencies listed explicitly:

```
% cat Construct
env = Environment()
env.Program('hello', 'hello.c')
% cat hello.c
#include "hello_string.h"
```

```
main()
{
        printf("%s\n", STRING);
}
% cat > hello_string.h
#define STRING  "Hello, world!\n"
% scons .
gcc -c hello.c -o hello.o
gcc -o hello hello.c
% ./hello
Hello, world!
% cat > hello_string.h
#define STRING  "Hello, world, hello!\n"
% scons .
gcc -c hello.c -o hello.o
gcc -o hello hello.c
% ./hello
Hello, world, hello!
%
```

Ignoring dependencies

Undesirable `automatic dependencies` or `implicit dependencies` may be ignored:

```
env.Program(target = 'bar', source = 'bar.c')
env.Ignore('bar', '/usr/bin/gcc', 'version.h')
```

In the above example, the `bar` program will not be rebuilt if the `/usr/bin/gcc` compiler or the `version.h` file change.

Explicit dependencies

Dependencies that are unknown to `SCons` may be specified explicitly in an `SCons` configuration file:

```
env.Dependency(target = 'output1', dependency = 'input_1 input_2')
env.Dependency(target = 'output2', dependency = ['input_1', 'input_2'])
env.Dependency(target = 'output3', dependency = ['white space input'])

env.Dependency(target = 'output_a output_b', dependency = 'input_3')
env.Dependency(target = ['output_c', 'output_d'], dependency = 'input_4')
env.Dependency(target = ['white space output'], dependency = 'input_5')
```

Just like the `target` keyword argument, the `dependency` keyword argument may be specified as a string of white-space separated file names, or as an array.

A dependency on an `SCons` configuration file itself may be specified explicitly to force a rebuild whenever the configuration file changes:

```
env.Dependency(target = 'archive.tar.gz', dependency = 'SConstruct')
```

`Scanner` Objects

Analagous to the previously-described `Builder` objects, `SCons` supplies (and uses) `Scanner` objects to search the contents of a file for implicit dependency files:

CScan	scan .{c,C,cc,cxx,cpp} files for #include dependencies

A `construction environment` can be explicitly initialized with associated `Scanner` objects:

```
env = Environment(SCANNERS = ['CScan', 'M4Scan'])
```

`Scanner` objects bound to a `construction environment` can be associated directly with specified files:

```
env.CScan('foo.c', 'bar.c')
env.M4Scan('input.m4')
```

User-defined `Scanner` objects

A user may define a `Scanner` object to scan a type of file for implicit dependencies:

```
def scanner1(file_contents):
        # search for dependencies
        return dependency_list

FirstScan = Scanner(function = scanner1)
```

The scanner function must return a list of dependencies that its finds based on analyzing the file contents it is passed as an argument.

The scanner function, when invoked, will be passed the calling environment. The scanner function can use `construction environments` from the passed environment to affect how it performs its dependency scan--the canonical example being to use some sort of search-path construction variable to look for dependency files in other directories:

```
def scanner2(file_contents, env):
        path = env.{'SCANNERPATH'} # XXX
        # search for dependencies using 'path'
        return dependency_list

SecondScan = Scanner(function = scanner2)
```

The user may specify an additional argument when the `Scanner` object is created. When the scanner is invoked, the additional argument will be passed to the scanner funciton, which can be used in any way the scanner function sees fit:

```
def scanner3(file_contents, env, arg):
        # skip 'arg' lines, then search for dependencies
        return dependency_list

Skip_3_Lines_Scan = Scanner(function = scanner2, argument = 3)
Skip_6_Lines_Scan = Scanner(function = scanner2, argument = 6)
```

Copying `Scanner` Objects

A method exists to return a copy of an existing `Scanner` object, with any overridden values specified as keyword arguments to the method:

```
scan = Scanner(function = my_scan)
scan_path = scan.Copy(path = '%SCANNERPATH')
```

Typically, `Scanner` objects will be supplied by a tool-master or administrator through a shared `construction environment`.

`Scanner` maps

* *If the `BUILDERMAP` proves unnecessary, we could/should get rid of this one, too, by adding a parallel `src_suffix` argument to the `Scanner` factory... Comments?*

Each `construction environment` has a `SCANNERMAP`, a dictionary that associates different file suffixes with a scanner object that can be used to generate a list of dependencies from the contents of that file. This `SCANNERMAP` can be initialized at instantiation:

```
env = Environment(SCANNERMAP = {
                        '.c'  : CScan,
                        '.cc' : CScan,
                        '.m4' : M4Scan,
                })
```

`Scanner` objects referenced in the `SCANNERMAP` do not need to be listed separately in the `SCANNERS` variable. The `construction environment` will bind the union of the `Scanner` objects listed in both variables.

Targets

The methods in the build engine API described so far merely establish associations that describe file dependencies, how a file should be scanned, etc. Since the real point is to actually *build* files, `SCons` also has methods that actually direct the build engine to build, or otherwise manipulate, target files.

Building targets

One or more targets may be built as follows:

```
env.Build(target = ['foo', 'bar'])
```

Note that specifying a directory (or other collective object) will cause all subsidiary/dependent objects to be built as well:

```
env.Build(target = '.')

env.Build(target = 'builddir')
```

By default, `SCons` explicitly removes a target file before invoking the underlying function or command(s) to build it.

Removing targets

A "cleanup" operation of removing generated (target) files is performed as follows:

```
env.Clean(target = ['foo', 'bar'])
```

Like the `Build` method, the `Clean` method may be passed a directory or other collective object, in which case the subsidiary target objects under the directory will be removed:

```
env.Clean(target = '.')

env.Clean(target = 'builddir')
```

(The directories themselves are not removed.)

Suppressing build-target removal

As mentioned, by default, `SCons` explicitly removes a target file before invoking the underlying function or command(s) to build it. Files that should not be removed before rebuilding can be specified via the `Precious` method:

```
env.Library(target = 'libfoo.a', source = ['aaa.c', 'bbb.c', 'ccc.c'])
env.Precious('libfoo.a')
```

Default targets

The user may specify default targets that will be built if there are no targets supplied on the command line:

```
env.Default('install', 'src')
```

Multiple calls to the `Default` method (typically one per `SConscript` file) append their arguments to the list of default targets.

File installation

Files may be installed in a destination directory:

```
env.Install('/usr/bin', 'program1', 'program2')
```

Files may be renamed on installation:

```
env.InstallAs('/usr/bin/xyzzy', 'xyzzy.in')
```

Multiple files may be renamed on installation by specifying equal-length lists of target and source files:

```
env.InstallAs(['/usr/bin/foo', '/usr/bin/bar'],
              ['foo.in', 'bar.in'])
```

Target aliases

In order to provide convenient "shortcut" target names that expand to a specified list of targets, aliases may be established:

```
env.Alias(alias = 'install',
          targets = ['/sbin', '/usr/lib', '/usr/share/man'])
```

In this example, specifying a target of `install` will cause all the files in the associated directories to be built (that is, installed).

An `Alias` may include one or more other `Aliases` in its list:

```
env.Alias(alias = 'libraries', targets = ['lib'])
env.Alias(alias = 'programs', targets = ['libraries', 'src'])
```

Customizing output

* *Take this whole section with a grain of salt. I whipped it up without a great deal of thought to try to add a "competitive advantage" for the second round of the Software Carpentry contest. In particular, hard-coding the analysis points and the keywords that specify them feels inflexible, but I can't think of another way it would be done effectively. I dunno, maybe this is fine as it is...*

The `SCons` API supports the ability to customize, redirect, or suppress its printed output through user-defined functions. `SCons` has several pre-defined points in its build process at which it calls a function to (potentially) print output. User-defined functions can be specified for these call-back points when `Build` or `Clean`is invoked:

```
env.Build(target = '.',
       on_analysis = dump_dependency,
       pre_update = my_print_command,
       post_update = my_error_handler)
       on_error = my_error_handler)
```

The specific call-back points are:

`on_analysis`

Called for every object, immediately after the object has been analyzed to see if it's out-of-date. Typically used to print a trace of considered objects for debugging of unexpected dependencies.

`pre_update`

Called for every object that has been determined to be out-of-date before its update function or command is executed. Typically used to print the command being called to update a target.

`post_update`

Called for every object after its update function or command has been executed. Typically used to report that a top-level specified target is up-to-date or was not remade.

`on_error`

Called for every error returned by an update function or command. Typically used to report errors with some string that will be identifiable to build-analysis tools.

Functions for each of these call-back points all take the same arguments:

```
my_dump_dependency(target, level, status, update, dependencies)
```

where the arguments are:

`target`

The target object being considered.

`level`

Specifies how many levels the dependency analysis has recursed in order to consider the `target`. A value of `0` specifies a top-level `target` (that is, one passed to the `Build` or `Clean` method). Objects which a top-level `target` is directly dependent upon have a `level` of <1>, their direct dependencies have a `level` of <2>, etc. Typically used to indent output to reflect the recursive levels.

`status`

A string specifying the current status of the target (`"unknown"`, `"built"`, `"error"`, `"analyzed"`, etc.). A complete list will be enumerated and described during implementation.

`update`

The command line or function name that will be (or has been) executed to update the `target`.

`dependencies`

A list of direct dependencies of the target.

Separate source and build trees

* *I've never liked Cons' use of the name `Link` for this functionality, mainly because the term is overloaded with linking object files into an executable. Yet I've never come up with anything better. Any suggestions?*

* *Also, I made this an `Environment` method because it logically belongs in the API reference (the build engine needs to know about it), and I thought it was clean to have everything in the build-engine API be called through an `Environment` object. But `Link` isn't really associated with a specific environment (the `Cons` classic implementation just leaves it as a bare function call), so maybe we should just follow that example and not call it through an environment...*

`SCons` allows target files to be built completely separately from the source files by "linking" a build directory to an underlying source directory:

```
env.Link('build', 'src')

SConscript('build/SConscript')
```

`SCons` will copy (or hard link) necessary files (including the `SConscript` file) into the build directory hierarchy. This allows the source directory to remain uncluttered by derived files.

Variant builds

The `Link` method may be used in conjunction with multiple `construction environments` to support variant builds. The following `SConstruct` and

`SConscript` files would build separate debug and production versions of the same program side-by-side:

```
% cat SConstruct
env = Environment()
env.Link('build/debug', 'src')
env.Link('build/production', 'src')
flags = '-g'
SConscript('build/debug/SConscript', Export(env))
flags = '-O'
SConscript('build/production/SConscript', Export(env))
% cat src/SConscript
env = Environment(CCFLAGS = flags)
env.Program('hello', 'hello.c')
```

The following example would build the appropriate program for the current compilation platform, without having to clean any directories of object or executable files for other architectures:

```
% cat SConstruct
build_platform = os.path.join('build', sys.platform)
Link(build_platform, 'src')
SConscript(os.path.join(build_platform, 'SConscript'))
% cat src/SConscript
env = Environment
env.Program('hello', 'hello.c')
```

Code repositories

* *Like `Link`, `Repository` and `Local` are part of the API reference, but not really tied to any specific environment. Is it better to be consistent about calling everything in the API through an environment, or to leave these independent so as not to complicate their calling interface?*

`SCons` may use files from one or more shared code repositories in order to build local copies of changed target files. A repository would typically be a central directory tree, maintained by an integrator, with known good libraries and executables.

```
Repository('/home/source/1.1', '/home/source/1.0')
```

Specified repositories will be searched in-order for any file (configuration file, input file, target file) that does not exist in the local directory tree. When building a local target file, `SCons` will rewrite path names in the build command to use the necessary repository files. This includes modifying lists of `-I` or `-L` flags to specify an appropriate set of include paths for dependency analysis.

`SCons` will modify the Python `sys.path` variable to reflect the addition of repositories to the search path, so that any imported modules or packages necessary for the build can be found in a repository, as well.

If an up-to-date target file is found in a code repository, the file will not be rebuilt or copied locally. Files that must exist locally (for example, to run tests) may be specified:

```
Local('program', 'libfoo.a')
```

in which case `SCons` will copy or link an up-to-date copy of the file from the appropriate repository.

Derived-file caching

* *There should be extensions to this part of the API for auxiliary functions like cleaning the cache.*

`SCons` can maintain a cache directory of target files which may be shared among multiple builds. This reduces build times by allowing developers working on a project together to share common target files:

```
Cache('/var/tmp/build.cache/i386')
```

When a target file is generated, a copy is added to the cache. When generating a target file, if `SCons` determines that a file that has been built with the exact same dependencies already exists in the specified cache, `SCons` will copy the cached file rather than re-building the target.

Command-line options exist to modify the `SCons` caching behavior for a specific build, including disabling caching, building dependencies in random order, and displaying commands as if cached files were built.

Job management

* *This has been completely superseded by the more sophisticated `Task` manager that Anthony Roach has contributed. I need to write that up...*

A simple API exists to inform the Build Engine how many jobs may be run simultaneously:

```
Jobs(limit = 4)
```

Notes

1. It would be nice if we could avoid re-inventing the wheel here by using some other Python-based tool `Autoconf` replacement--like what was supposed to come out of the Software Carpentry configuration tool contest. It will probably be most efficient to roll our own logic initially and convert if something better does come along.

Chapter 5. Native Python Interface

The "Native Python" interface is the interface that the actual `SCons` utility will present to users. Because it exposes the Python Build Engine API, `SCons` users will have direct access to the complete functionality of the Build Engine. In contrast, a different user interface such as a GUI may choose to only use, and present to the end-user, a subset of the Build Engine functionality.

Configuration files

`SCons` configuration files are simply Python scripts that invoke methods to specify target files to be built, rules for building the target files, and dependencies. Common build rules are available by default and need not be explicitly specified in the configuration files.

By default, the `SCons` utility searches for a file named `SConstruct`, `Sconstruct` or `sconstruct` (in that order) in the current directory, and reads its configuration from the first file found. A `-f` command-line option exists to read a different file name.

Python syntax

Because `SCons` configuration files are Python scripts, normal Python syntax can be used to generate or manipulate lists of targets or dependencies:

```
sources = ['aaa.c', 'bbb.c', 'ccc.c']
env.Make('bar', sources)
```

Python flow-control can be used to iterate through invocations of build rules:

```
objects = ['aaa.o', 'bbb.o', 'ccc.o']
for obj in objects:
        src = replace(obj, '.o', '.c')
        env.Make(obj, src)
```

or to handle more complicated conditional invocations:

```
# only build 'foo' on Linux systems
if sys.platform == 'linux1':
        env.Make('foo', 'foo.c')
```

Because `SCons` configuration files are Python scripts, syntax errors will be caught by the Python parser. Target-building does not begin until after all configuration files are read, so a syntax error will not cause a build to fail half-way.

Subsidiary configuration Files

A configuration file can instruct `SCons` to read up subsidiary configuration files. Subsidiary files are specified explicitly in a configuration file via the `SConscript` method. As usual, multiple file names may be specified with white space separation, or in an array:

```
SConscript('other_file')
SConscript('file1 file2')
SConscript(['file3', 'file4'])
SConscript(['file name with white space'])
```

An explicit `sconscript` keyword may be used:

```
SConscript(sconscript = 'other_file')
```

Including subsidiary configuration files is recursive: a configuration file included via `SConscript` may in turn `SConscript` other configuration files.

Variable scoping in subsidiary files

When a subsidiary configuration file is read, it is given its own namespace; it does not have automatic access to variables from the parent configuration file.

Any variables (not just `SCons` objects) that are to be shared between configuration files must be explicitly passed in the `SConscript` call using the `Export` method:

```
env = Environment()
debug = Environment(CCFLAGS = '-g')
installdir = '/usr/bin'
SConscript('src/SConscript', Export(env=env, debug=debug, installdir=installdir))
```

* *The `env=env` stuff bugs me because it imposes extra work on the normal case where you don't rename the variables. Can we simplify the `Export` method so that a string without a keyword assignment is split into variables that are passed through transparently? Equivalent to the above example: `SConscript('src/SConscript', Export('env debug installdir'))`*

Which may be specified explicitly using a keyword argument:

```
env = Environment()
debug = Environment(CCFLAGS = '-g')
installdir = '/usr/bin'
SConscript(sconscript = 'src/SConscript',
           export = Export(env=env, debug=debug, installdir=installdir))
```

Explicit variable-passing provides control over exactly what is available to a subsidiary file, and avoids unintended side effects of changes in one configuration file affecting other far-removed configuration files (a very hard-to-debug class of build problem).

Hierarchical builds

The `SConscript` method is so named because, by convention, subsidiary configuration files in subdirectories are named `SConscript`:

```
SConscript('src/SConscript')
SConscript('lib/build_me')
```

When a subsidiary configuration file is read from a subdirectory, all of that configuration file's targets and build rules are interpreted relative to that directory (as if `SCons` had changed its working directory to that subdirectory). This allows for easy support of hierarchical builds of directory trees for large projects.

Sharing construction environments

SCons will allow users to share construction environments, as well as other SCons objects and Python variables, by importing them from a central, shared repository using normal Python syntax:

```
from LocalEnvironments import optimized, debug

optimized.Make('foo', 'foo.c')
debug.Make('foo-d', 'foo.c')
```

The expectation is that some local tool-master, integrator or administrator will be responsible for assembling environments (creating the Builder objects that specify the tools, options, etc.) and make these available for sharing by all users.

The modules containing shared construction environments (LocalEnvironments in the above example) can be checked in and controlled with the rest of the source files. This allows a project to track the combinations of tools and command-line options that work on different platforms, at different times, and with different tool versions, by using already-familiar revision control tools.

Help

The SCons utility provides a Help function to allow the writer of a SConstruct file to provide help text that is specific to the local build tree:

```
Help("""
Type:
        scons .         build and test everything
        scons test      build the software
        scons src       run the tests
        scons web       build the web pages
""")
```

This help text is displayed in response to the -h command-line option. Calling the Help function more than once is an error.

Debug

SCons supports several command-line options for printing extra information with which to debug build problems.

* *These need to be specified and explained beyond what the man page will have.*

See the -d, -p, -pa, and -pw options in the , below. All of these options make use of call-back functions to printed by the Build Engine.

Chapter 6. Installation

* *THIS CHAPTER NEEDS TO BE DISCUSSED AND WRITTEN.*

Chapter 7. Other Issues

No build tools is perfect. Here are some `SCons` issues that do not yet have solutions.

Interaction with SC-config

The SC-config tool will be used in the `SCons` installation process to generate an appropriate default construction environment so that building most software works "out of the box" on the installed platform. The SC-config tool will find reasonable default compilers (C, C++, Fortran), linkers/loaders, library archive tools, etc. for specification in the default `SCons` construction environment.

Interaction with test infrastructures

`SCons` can be configured to use SC-test (or some other test tool) to provide controlled, automated testing of software. The `Link` method could link a `test` subdirectory to a build subdirectory:

```
Link('test', 'build')
SConscript('test/SConscript')
```

Any test cases checked in with the source code will be linked into the `test` subdirectory and executed. If `SConscript` files and test cases are written with this in mind, then invoking:

```
% sccons test
```

Would run all the automated test cases that depend on any changed software.

Java dependencies

Java dependencies are difficult for an external dependency-based construction tool to accomodate. Determining Java class dependencies is more complicated than the simple pattern-matching of C or C++ `#include` files. From the point of view of an external build tool, the Java compiler behaves "unpredictably" because it may create or update multiple output class files and directories as a result of its internal class dependencies.

An obvious `SCons` implementation would be to have the `Scanner` object parse output from **Java -depend -verbose** to calculate dependencies, but this has the distinct disadvantage of requiring two separate compiler invocations, thereby slowing down builds.

Limitations of digital signature calculation

In practice, calculating digital signatures of a file's contents is a more robust mechanism than time stamps for determining what needs building. However:

1. Developers used to the time stamp model of `Make` can initially find digital signatures counter-intuitive. The assumption that:

   ```
   % touch file.c
   ```

   will cause a rebuild of `file` is strong...
2. Abstracting dependency calculation into a single digital signature loses a little information: It is no longer possible to tell (without laborious additional calculation) which input file dependency caused a rebuild of a given target file. A

feature that could report, "I'm rebuilding file X because it's out-of-date with respect to file Y," would be good, but an digital-signature implementation of such a feature is non-obvious.

Remote execution

The ability to use multiple build systems through remote execution of tools would be good. This should be implementable through the `Job` class. Construction environments would need modification to specify build systems.

Conditional builds

The ability to check run-time conditions as suggested on the sc-discuss mailing list ("build X only if: the machine is idle / the file system has Y megabytes free space") would also be good, but is not part of the current design.

Chapter 8. Background

Most of the ideas in SCons originate with Cons, a Perl-based software construction utility that has been in use by a small but growing community since its development by Bob Sidebotham at FORE Systems in 1996. The Cons copyright was transferred in 2000 from Marconi (who purchased FORE Systems) to the Free Software Foundation. I've been a principal implementer and maintainer of Cons for several years.

Cons was originally designed to handle complicated software build problems (multiple directories, variant builds) while keeping the input files simple and maintainable. The general philosophy is that the build tool should "do the right thing" with minimal input from an unsophisticated user, while still providing a rich set of underlying functionality for more complicated software construction tasks needed by experts.

In 2000, the Software Carpentry sought entries in a contest for a new, Python-based build tool that would provide an improvement over Make for physical scientists and other non-programmers struggling to use their computers more effectively. Prior to that, the idea of combining the superior build architecture of Cons with the easier syntax of Python had come up several times on the cons-discuss mailing list. The Software Carpentry contest provided the right motivation to spend some actual time working on a design document.

After two rounds of competition, the submitted design, named ScCons, won the competition. Software Carpentry, however, did not immediately fund implementation of the build tool, instead contracting for additional, more detailed draft(s) of the design document. This proved to be not as strong motivation as actual coding, and after several months of inactivity, I essentially resigned from the Software Carpentry effort in early 2001 to start working on the tool independently.

After half a year of prototyping some of the important infrastructure, I accumulated enough code to take the project public at SourceForge, renaming it SCons to distinguish it slightly from the version of the design that won the Software Carpentry contest while still honoring its roots there and in the original Cons utility. And also because it would be a teensy bit easier to type.

Chapter 9. Summary

SCons offers a robust and feature-rich design for an SC-build tool. With a Build Engine based on the proven design of the Cons utility, it offers increased simplification of the user interface for unsophisticated users with the addition of the "do-the-right-thing" env.Make method, increased flexibility for sophisticated users with the addition of Builder and Scanner objects, a mechanism to allow tool-masters (and users) to share working construction environments, and embeddability to provide reliable dependency management in a variety of environments and interfaces.

Chapter 10. Acknowledgements

I'm grateful to the following people for their influence, knowing or not, on the design of `SCons`:

Bob Sidebotham

> First, as the original author of `Cons`, Bob did the real heavy lifting of creating the underlying model for dependency management and software construction, as well as implementing it in Perl. During the first years of `Cons`' existence, Bob did a skillful job of integrating input and code from the first users, and consequently is a source of practical wisdom and insight into the problems of real-world software construction. His continuing advice has been invaluable.

The `SCons` Development Team

> A big round of thanks go to those brave souls who have gotten in on the ground floor: David Abrahams, Charles Crain, Steven Leblanc. Anthony Roach, and Steven Shaw. Their contributions, through their general knowledge of software build issues in general Python in particular, have made `SCons` what it is today.

The `Cons` Community

> The real-world build problems that the users of `Cons` share on the **cons-discuss** mailing list have informed much of the thinking that has gone into the `SCons` design. In particular, Rajesh Vaidheeswarran, the current maintainer of `Cons`, has been a very steady influence. I've also picked up valuable insight from mailing-list participants Johan Holmberg, Damien Neil, Gary Oberbrunner, Wayne Scott, and Greg Spencer.

Peter Miller

> Peter has indirectly influenced two aspects of the `SCons` design:
>
> Miller's influential paper *Recursive Make Considered Harmful* was what led me, indirectly, to my involvement with `Cons` in the first place. Experimenting with the single-Makefile approach he describes in *RMCH* led me to conclude that while it worked as advertised, it was not an extensible scheme. This solidified my frustration with Make and led me to try `Cons`, which at its core shares the single-process, universal-DAG model of the "RMCH" single-Makefile technique.
>
> The testing framework that Miller created for his Aegis change management system changed the way I approach software development by providing a framework for rigorous, repeatable testing during development. It was my success at using Aegis for personal projects that led me to begin my involvement with `Cons` by creating the **cons-test** regression suite.

Stuart Stanley

> An experienced Python programmer, Stuart provided valuable advice and insight into some of the more useful Python idioms at my disposal during the original `ScCons`; design for the Software Carpentry contest.

Gary Holt

> I don't know which came first, the first-round Software Carpentry contest entry or the tool itself, but Gary's design for `Make++` showed me that it is possible to marry the strengths of `Cons`-like dependency management with backwards compatibility for `Makefile`s. Striving to support both `Makefile` compatibility and a native Python interface cleaned up the `SCons` design immeasurably by factoring out the common elements into the Build Engine.